RuleWizard User's Guide

Introducing RuleWizard

Creating Custom Coding Standards

RuleWizard GUI Help

Reference Guide

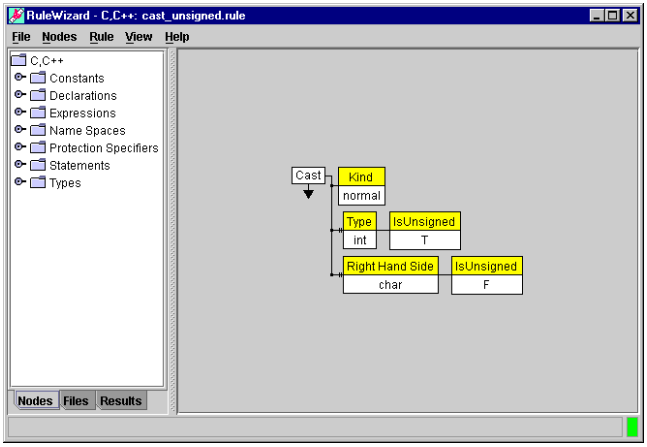
Index

Welcome!

Welcome to RuleWizard, a feature that allows you to create custom coding standards. CodeWizard can automatically enforce any rule created in RuleWizard.

By allowing you to easily create and enforce standards that are perfectly tailored to your personal, project, team, and company needs, RuleWizard and CodeWizard provide the most effective and efficient error prevention solution available.

With RuleWizard, you create custom rules by graphically expressing the pattern that you want CodeWizard to look for when it parses code during static analysis. Rules are created by selecting a main "node," then adding additional elements until the rule expresses the pattern that you want CodeWizard to check for. Rule elements are added by pointing, clicking, and entering values into dialog boxes. No knowledge of the parser is required to write or modify a rule.



Contacting ParaSoft

ParaSoft is committed to providing you with the best possible product support for RuleWizard. If you have any trouble installing or using RuleWizard, please follow the procedure below in contacting our Quality Consulting department.

- Check the manual.
- Be prepared to recreate your problem.
- Know your RuleWizard version. (You can find it in **Help> About**).
- If the problem is not urgent, report it by e-mail or by fax.
- If you call, please use a phone near your computer. The Quality Consultant may need you to try things while you are on the phone.

Contact Information

- **USA Headquarters**

  Tel: (888) 305-0041

  Fax: (626) 305-9048

  Email: quality@parasoft.com

  Web Site: http://www.parasoft.com

- **ParaSoft France**

  Tel: +33 (0) 1 60 79 51 51

  Fax: +33 (0) 1 60 79 51 50

  Email: quality@parasoft-fr.com

- **ParaSoft Germany**

  Tel: +49 (0) 78 05 95 69 60

  Fax: +49 (0) 78 05 95 69 19

  Email: quality@parasoft-de.com

- **ParaSoft UK**

  Tel: +44 171 288 66 00

  Fax: +44 171 288 66 02

  Email: quality@parasoft-uk.com

Overview: How to Create a Rule

When you create a rule, your goal is to graphically express the pattern that you *do not* want to appear in your code. When CodeWizard enforces a rule, it searches for instances where the specified pattern occurs, then flags any violations that it finds.

To open RuleWizard, click the RuleWizard button in the CodeWizard Visual C++ tool bar (for Windows) or enter `rulewizard` at the prompt (for UNIX).

When the RuleWizard GUI opens, you will see two main panels. The Node tab on the left side of the GUI contains the nodes that you can use as rule subjects.The right side of the GUI is the area where your rule patterns will be displayed.

Creating a rule generally involves the following steps:

1. Using plain English, define the rule that you want to create and enforce.

   For example: "Begin class names with an uppercase letter."

2. Express this concept in terms of RuleWizard elements.

   - Create the parent rule node that is the subject of your rule.

     a. In the Node tab, select the node that is the subject of your rule.

        Tip: For a description of a certain node, right click the node, and choose **View Documentation** from the shortcut menu that opens.

     b. Right-click the selected node, then choose **Create Rule** from the shortcut menu. A rule node will appear in the GUI's right panel.

   - Add further qualifications to your node until it fully expresses your rule.

a. Right-click any of your rule's nodes. All available options for the chosen node will be displayed in the shortcut menu that opens. Any options that are not programming elements or concepts are explained in the RuleWizard Commands topic of this User's Guide.

b. Choose a command from the shortcut menu. Depending on the command that you choose, RuleWizard will add a rule element, modify a rule element, or open a dialog box that lets you add or modify a rule element.

c. If you want to continue adding to and modifying the rule, you can do so by right-clicking any rule node or rule element, then choosing one of the available commands.

- Determine what error message will be displayed when this rule is violated.

  a. Create an output arrow by right-clicking a rule node (the placement of the output arrow determines what line number is used for the output message; to have the line number of node C included in the output message, place the output arrow on node C), then choosing **Create Output** from the shortcut menu. The Customize Output window will open.

  b. In the Customize Output window, enter a brief explanation of the violation.

  c. Click **OK**.
  An output arrow will then be added to your rule.

The rule is complete once it:

- Expresses the pattern that you want CodeWizard to search for, and
- Contains an output arrow and message.

After you customize this rule's properties (you must at least enter a rule header) and save the rule, you can enforce it with CodeWizard.

Information about the rule will be included in RuleWizard's Rules Documentation files.To view this documentation, choose **Help> Rules Documentation**. To refresh this documentation, choose **Rule> Update Documentation**.

Tips

General

- Remember that you are trying to express a pattern that constitutes a violation of the rule.
- As you create your rule, look at the status bar for tips on creating a valid rule. The color of the bar in the right side of the status bar indicates whether or not a rule is valid: a red bar indicates that the rule is not yet valid; a green bar indicates that the rule is valid. The messages in the status bar tell you how to make an invalid rule valid.
- Be sure to include at least one output arrow in each rule. If a rule does not have an output arrow, CodeWizard will not report an error if this pattern is found in the code under test. To include an output arrow, right-click a rule node (the placement of the output arrow determines what line number is used for the output message; to have the line number of node C included in the output message, place the output arrow on node C), then choose **Create Output** from the shortcut menu.
- Be sure to enter a header when you customize this rule's properties; rules without headers are not valid.
- Both nodes (such as **bool Constant**) and folders containing nodes (such as **Constants**) can be used as rule nodes.
- To view a description of a node in the Node tab, right-click the node that you want more information about, then choose **View Documentation** from the shortcut menu.

- Be aware that RuleWizard is order-specific. If your rule has more than one child node, you can move the child up or down one position by right-clicking the vertical line common to all children, then choosing **Move up one** or **Move down one** from the shortcut menu.

Expressions and Regular Expressions

- Use expressions to match values, and regular expressions to match strings. For guidelines on using expressions and regular expressions, see “Expressions and Regular Expressions” on page 63.

Rule Conditions

- To create a rule that whose pattern involves a union, intersection, or difference of multiple nodes, see “Working With Node Sets” on page 32.
- To create a rule condition that restricts the number of a certain element that can appear in a block (like a file or a class), create a collector to track the number of instances of that pattern, then use **Count** to specify the number of instances that constitutes a violation. For information on determining exactly how “counts” are calculated, see “Working With Node Sets” on page 32.
- To create a rule condition about the code element that contains the parent node, use **Context**.
- To create a rule condition about the parent node’s condition statement, use **Condition**.
- To create a rule condition about the code element that is a subnode of the parent node, use **Body**.
- To indicate whether or not CodeWizard searches nodes recursively when it searches for the rule condition, use **Indirect Check** and **Direct Check**.
    - **Indirect Check** is the default setting; to use a **Direct Check**, right-click the node whose checking type you want to change, then choose **Direct Check** from the shortcut menu.

How to Customize Rule Properties

To specify such rule properties as Rule ID, header, severity, author,and description, choose **Rule> Properties**, then enter the desired properties in the Rule Properties panel.

How to Save a Rule

In order to enforce a rule, you must save it. To save your rule, choose **Rule> Save** or **Rule> Save As**. This command will invoke a file chooser in which you can specify the rule's filename and path. Be sure to give each rule a .rule extension. If you do not use this exact extension,CodeWizard will not load your rules properly.

After you have saved your rule(s), you can exit RuleWizard, then enable your rule.

You can determine if RuleWizard automatically enables saved rules without asking, if it asks you about enforcing each rule you save, or if it does not enforce saved rules by choosing one of the options available in the RuleWizard Preferences panel.

How to Automatically Enforce Your Custom Coding Standards

To have CodeWizard enforce an enabled coding standard, run CodeWizard as normal.

If you want CodeWizard to enforce a rule that you have not yet enabled, perform the following steps:

UNIX

1. If the rule that you want to enforce is saved in the default directory (<codewizard_install_dir>/rules), open the cwrules.txt file in that directory.

   If the rule that you want to enforce is saved in a different directory, create a text file in that directory.

2. In the open text file, add the names of the rule files (i.e. IfAssign.rule) that you want CodeWizard to enforce. When you add the rule files' names, do not use any spaces or indentation. Add only one rule file's name per line.

3. Save the text file.

4. If your text file is not <codewizard_install_dir>/rules/cwrules.txt, open your .psrc file, enter a rulesdirlist option that specifies the path to the text file you just saved, then save your .psrc file.

   For example, if your rules and text file named rules.txt were saved in /home/user/rules, you would enter

```
CodeWizard.ruledirlist /home/user/rules/rules.txt
```

5. Run CodeWizard as normal to automatically enforce these rules.

Windows

1. If your rules are saved in the default directory (<codewizard_install_dir/rules>), proceed to step 2.

If your rules are stored in a different directory, perform the following steps:

- Create an empty text file in the directory that contains the rule(s) that you want to enforce.
- Save the text file.
- Open the CodeWizard Control Panel.
- Click the **Rules** Tab.
- Click **Add**, then specify the path to the text file that you just saved.
- Click **Apply**.
- Proceed to step 4.

2. Open the CodeWizard Control Panel.
3. Click the **Rules** Tab.
4. Check the check box(es) associated with the rule(s) that you want CodeWizard to enforce.
5. Click **Apply** and/or **OK**.
6. Run CodeWizard as normal to automatically enforce these rules.

Suppressing and Disabling Rules

If a custom rule is not relevant to a particular situation, you may want to suppress reporting of that rule's violations. Rules created in RuleWizard can be suppressed in the same way that built-in CodeWizard rules are suppressed. For information about suppressions, refer to the CodeWizard User's Guide. When suppressing custom rules, be aware that these rules are categorized as User [RuleID] (Rule ID is specified in the Rule Properties panel).

When you suppress a rule, CodeWizard checks for violations, but does not report them. If you do not want violations of a particular rule reported under most circumstances, you may improve testing performance by disabling the rule, then enabling the rule only when you want to enforce it.

If you never enabled a rule, the rule is already disabled.

To disable a previously enabled rule, perform the following steps:

UNIX

1. Open the text file that tells CodeWizard to enforce that rule.
2. In that text file, remove (or comment out) the names of the rule file (i.e. IfAssign.rule) that you do not want enforced.
3. Save the text file.

Windows

1. Open the CodeWizard Control Panel.
2. Click the Rules Tab.
3. In the Rules Directory List, select the entry for the file that tells CodeWizard to enforce this rule.
4. Clear the check box associated with the rule that you want to disable.
5. Click **Apply** and/or **OK**.

Enforcing Multiple Rule Sets

In some cases, you may want to have multiple rule sets. For example, you might want a company set, a team set, and a project set.

To create such a rule set:

1. In the directory where the rules are saved, create a text file that contains the names of the rule files (i.e. IfAssign.rule) that you want enforced. When you add the rule files' names, do not use any spaces or indentation. Add only one rule file's name per line.
2. Save that text file.
3. Tell CodeWizard which rule set(s) you want it to enforce by performing the appropriate procedure below.

UNIX

1. In your .psrc file, enter the name and path of the rule set(s) that you want to enforce.

- For example, if you wanted to enforce /directory1/site_rules.txt, /directory2/proj_rules.txt, and /directory3/user_rules.txt, you would enter

```
CodeWizard.ruledirlist /directory1/site_rules.txt
CodeWizard.ruledirlist /directory2/proj_rules.txt
CodeWizard.ruledirlist /directory3/user_rules.txt
```

- If all three lines are in a .psrc file, all the rules listed in these three files will be enforced.
- If you want to temporarily disable a rule set, comment out the line that contains the rule set that you want to skip.

2. Save your .psrc file.
3. Run CodeWizard as normal.

Windows

1. Open the CodeWizard Control Panel.
2. Click the **Rules** Tab.
3. Click **Add**, then specify the path to one of the text files that contains a rule set that you want to enforce.
4. If you want to add additional rule sets, repeat the previous step for all additional rule sets.
5. If you do not want to enforce a rule set listed in the Rules Directory List, select that entry, then click **Remove**.
6. Click **Apply**.
7. Run CodeWizard as normal.

Tutorial: Creating and Enforcing Custom Coding Standards

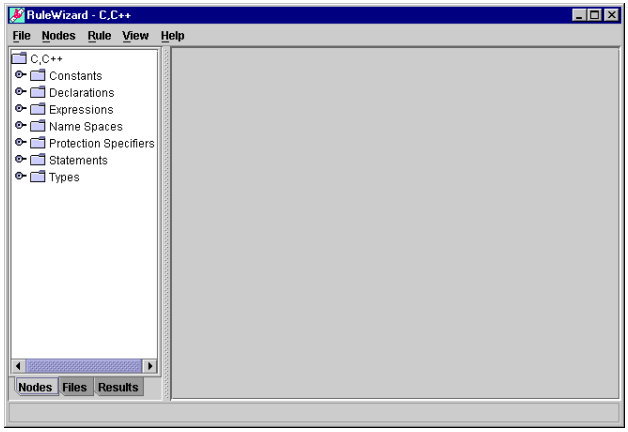
This tutorial will walk you through the steps required to compose and automatically enforce two basic rules:

1. Do not use the ?: operator (Lesson 1)
2. Function names should begin with a capital letter (Lesson 2)

Lesson 1: Do not use the ?: operator

Getting Started

When you first open Rule Wizard, you will see the following GUI:



The nodes with which you build your rules are displayed on the left pane of the GUI. The gray pane on the right of the GUI will be your workspace for composing rules.

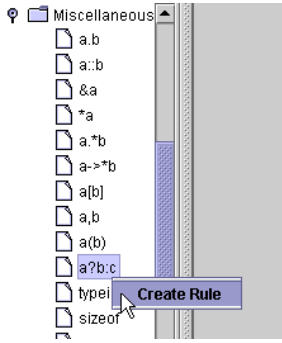
Creating the Rule

Let's start by walking you through the steps required to build a simple rule: Do not use the ?: operator.

Designing the Rule Pattern

Creating a Parent Rule Node

Whenever you create a rule, the first thing that you need to do is right-click the node that you want to be the subject of your rule, then select **Create Rule** from the shortcut menu. To begin creating this rule, right-click the **a?b:c** node in (under **Expressions> Miscellaneous**), then choose **Create Rule** from the shortcut menu.

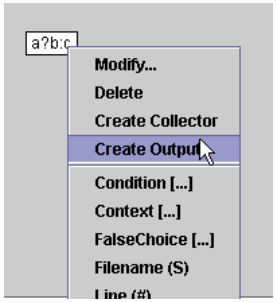


Adding Further Qualifications to the Parent Rule Node

The rule already expresses the pattern that we want CodeWizard to search for (the presence of the ?: operator in a file). Thus, no further qualifications are required.

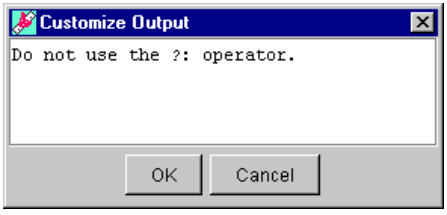
Specifying an Error Message

Now that your rule fully expresses the pattern, you need to specify what text CodeWizard should print when this rule is violated. The first step in doing this is right-clicking the parent rule node (here, the **a?b:c** rule node), then choosing **Create Output** from the shortcut menu.

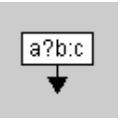


You will then see the following Customize Output window:

In the Customize Output window that appears, type "Do not use the ?: operator".



Next, click **OK**. Your rule should now look like this:

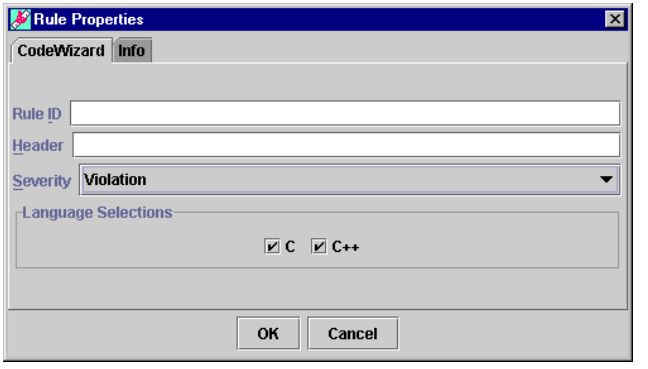


Your rule now tells CodeWizard to report the specified error message when the ?: operator is used. Your rule is now complete. After you customize this rule's properties and save it, CodeWizard will be able to enforce it.

Customizing Rule Properties

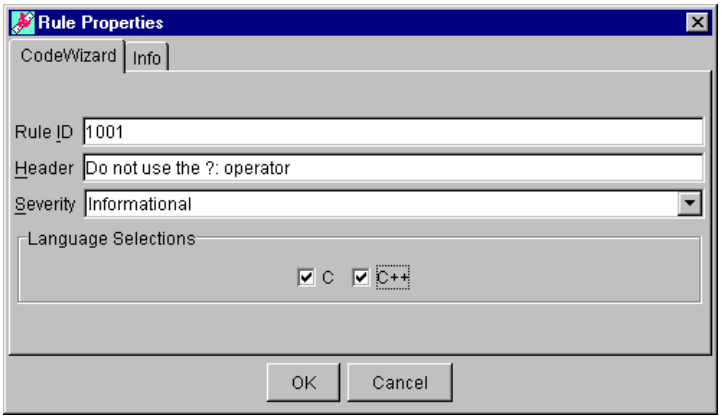
You can customize rule properties in the Rule Properties panel. To access this panel, choose **Rule> Properties**.

You will then see the Rule Properties panel.



The Rule Properties panel allows you to determine the rule's properties. In the CodeWizard tab, type the Rule ID (the number that you want CodeWizard to assign to this rule), the header (the name that you want CodeWizard to assign to this rule), then choose the rule's severity (the

severity category in which CodeWizard will classify the rule). For this rule, you might enter the following properties in the CodeWizard tab:



Now click the **Info** tab, then enter the name of the rule's author (your name and/or development group) and a description of the rule (for example, "C++ Style Sheet").

When you have entered all of these values, click **OK** to close this panel.

For more information on any of the fields in the Rule Properties panel, see the Rule Properties Panel topic.

Saving and Enabling Your Rule

Before you begin composing another rule, or before you exit the program, you will want to save your rule (CodeWizard only enforces rules that have been saved).

To tell RuleWizard to automatically enforce all rules that you save, choose **File> Customize Preferences**, click the **Rule Files** tab, then choose the **Enable rules automatically; do not ask me** option. This will prompt RuleWizard to automatically enable all rules that you save.

To save and enable your rule, choose **Rule> Save** or **Rule> Save As**. This command will invoke a file chooser in which you can specify the rule's filename and path. Be sure to give each rule a .rule extension. If you do not use this exact extension, CodeWizard will not load your rules properly.

Viewing Rule Documentation

Information about the rule will be included in RuleWizard's Rules Documentation files.To view this documentation, choose **Help> Rules Documentation**. To refresh this documentation, choose **Rule> Update Documentation**.

Enforcing Your Rule Automatically

To have CodeWizard enforce an enabled custom coding standard, simply run CodeWizard as normal.

Lesson 2: Function Names Must Begin With a Capital Letter

We will now demonstrate how to build a rule that flags instances where function names do not begin with a capital letter.

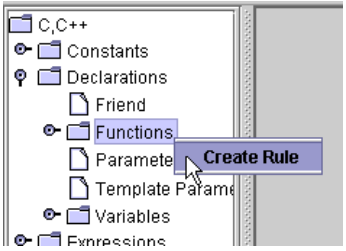
Designing the Rule Pattern

Creating the Parent Node

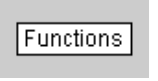
First, open RuleWizard (if it is not already open).

To start creating this rule, open **Declarations**, right-click the **Functions** folder, then choose **Create Rule** from the shortcut menu.

Note: Because we want this rule to apply to both member and global function, we selected the general **Function** node, not **Global Functions** or **Member Functions**.



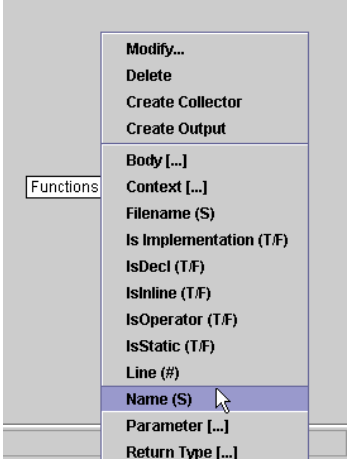
After you choose **Create Rule**, you will see the following parent rule node in the right pane of the GUI:



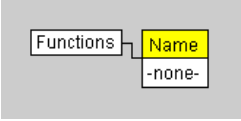
We now have the basic building block for a rule about some aspect of function declarations.

Adding Further Qualifications to the Parent Rule Node

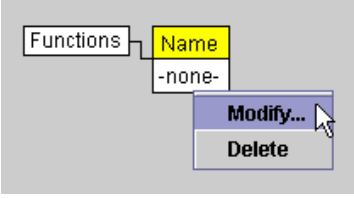
To specify that this rule will be about function naming conventions, right-click the **Functions** rule node and choose **Name** from the shortcut menu that opens.



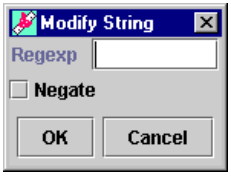
A rule node with the content **Name: -none-** will now be attached to your parent rule node.



To continue developing the rule, right-click the **Name** rule node, then choose **Modify** from the shortcut menu.



You will then see the following Modify String window:

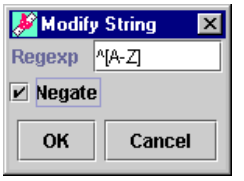


In the **Regexp** field of the Modify String Window, specify what value CodeWizard should look for.

- If you want CodeWizard to report an error if a certain value *is not* present in the function's name, (as we do in our example), enter the value that you want to require the presence of, then check the

Negate check box. This tells CodeWizard to report an error if the specified value is *not* present.

- In our example, we want the function name to begin with a capital letter. Thus, we would enter `^[A-Z]` in the **Regexp** field, then check the **Negate** check box:



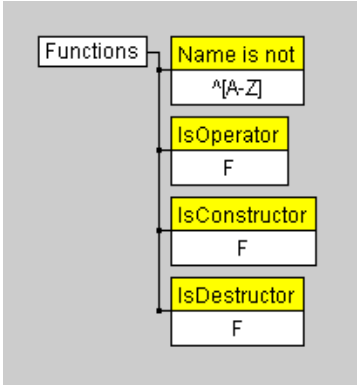
^ indicates the beginning of an expression; [A-Z] indicates uppercase letters from A to Z.

After you have typed this value and checked the check box, click **OK**.

Optional Additions

- **If you want to exempt operator functions from this rule:**
  1. Right-click the **Function** rule node.
  2. Choose **IsOperator** from the shortcut menu.
  3. Right-click the **IsOperator** rule node.
  4. Choose **Toggle** from the shortcut menu.
- **If you want to exempt constructor functions from this rule:**
  1. Right-click the **Function** rule node.
  2. Choose **Member Function> IsConstructor** from the shortcut menu.
  3. Right-click the **IsConstructor** rule node.
  4. Choose **Toggle** from the shortcut menu.
- **If you want to exempt destructor functions from this rule:**
  1. Right-click the **Function** rule node.

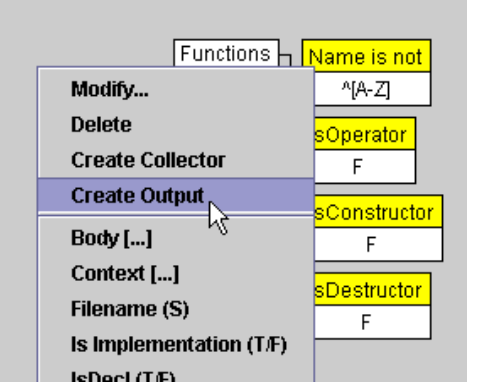
2. Choose **Member Function> IsDestructor** from the shortcut menu.
3. Right-click the **IsDestructor rule** node.
4. Choose **Toggle** from the shortcut menu.



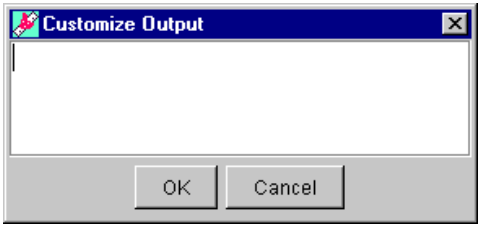
When you enter these conditions, CodeWizard will *not* check whether or not the names of operator functions, constructor functions, or destructor functions begin with a capital letter.

Specifying an Error Message

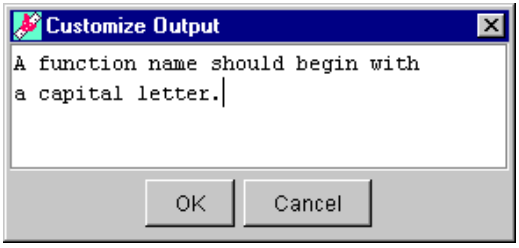
Finally, we need to specify what text CodeWizard should print when this rule is violated. The first step in doing this is right-clicking the parent rule node (here, the **Functions** rule node), then choosing **Create Output** from the shortcut menu.



The action will invoke the following Customize Output window:

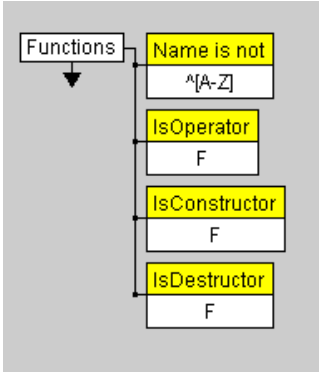


In the Customize Output window, enter the message that you want CodeWizard to deliver when this rule is violated. In this example, you might enter "A function name should begin with a capital letter."



Click **OK**.

Your rule should now look like this:

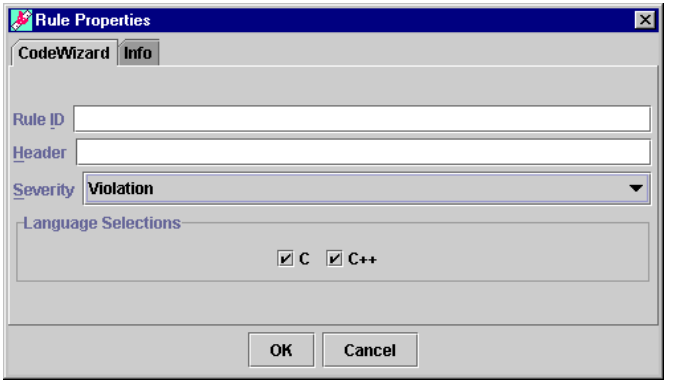


Your rule now tells CodeWizard to report the specified error message when a function's name does not begin with a capital letter. Your rule is now complete. After you customize this rule's properties and save it, CodeWizard will be able to enforce it.

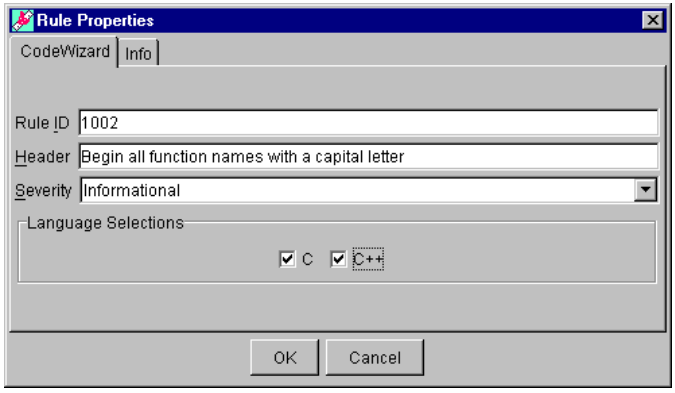
Customizing Rule Properties

Rule properties can be customized via the Rule Properties panel. To access this panel, choose **Rule> Properties**.

You will then see the Rule Properties panel.



This panel lets you determine the rule's properties. In the **CodeWizard** tab, type the Rule ID (the number that you want CodeWizard to assign to this rule), the header (the name that you want CodeWizard to assign to this rule), then choose the rule's severity (the severity category in which CodeWizard will classify the rule). For this rule, you might enter the following properties in the CodeWizard tab:



Click the **Info** tab, then enter the name of the rule's author (your name and/or development group) and a description of the rule (for example, "C++ Style Sheet"). Next, indicate that this rule is of Informational severity by choosing **Informational** from the Severity box

When you have entered all of these values, click **OK** to close this panel.

For more information on any of the fields in the Rule Properties panel, see the Rule Properties Panel topic.

Saving and Enabling Your Rule

Before you begin composing another rule, or before you exit the program, you will want to save your rule (CodeWizard only enforces rules that have been saved).

To tell RuleWizard to automatically enforce all rules that you save, choose **File> Customize Preferences**, click the **Rule Files** tab, then choose the **Enable rules automatically; do not ask me** option. This will prompt RuleWizard to automatically enable all rules that you save.

To save and enable your rule, choose **Rule> Save** or **Rule> Save As**. This command will invoke a file chooser in which you can specify the rule's filename and path. Be sure to give each rule a .rule extension. If you

do not use this exact extension, CodeWizard will not load your rules properly.

Viewing Rule Documentation

Information about the rule will be included in RuleWizard's Rules Documentation files.To view this documentation, choose **Help> Rules Documentation**. To refresh this documentation, choose **Rule> Update Documentation**.

Enforcing Your Rule Automatically

To have CodeWizard enforce an enabled custom coding standard, simply run CodeWizard as normal.

How to Modify an Existing Rule

RuleWizard can be used to modify two types of rules:

- All rules that you created can be modified in RuleWizard.
- CodeWizard rules that are in the “User” category, and have a corresponding .rule file.

  For more information about these rules, refer to the CodeWizard User’s Guide.

To modify a rule, perform the following steps:

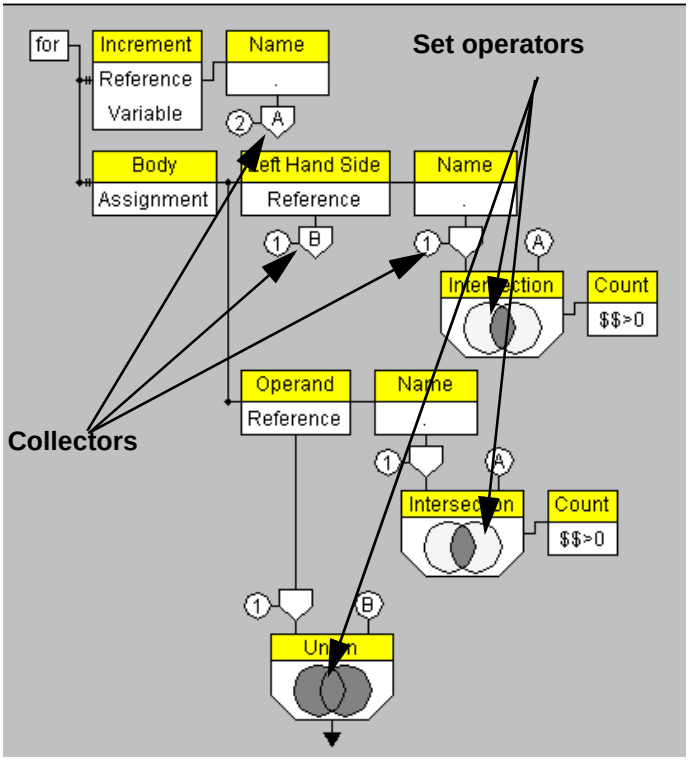
1. In RuleWizard, choose **Rule> Open**, then open the .rule file for the rule that you want to modify. By default, all .rule files for built-in user rules are located at <codewizard_install_dir>/rules.
2. Modify the rule until it fully expresses the pattern that you want CodeWizard to search for.
3. Save your rule by choosing **Rule> Save**.

Working With Node Sets

About Node Sets

When creating complex rules, you may want to create conditions that depend on specific attributes or relationships between multiple nodes. For example, if you have multiple nodes in a rule condition, you may want to specify exactly where CodeWizard starts and stops counting the number of “hits” (instances where the specified conditions are met). Or, you may want a rule to check if the total number of “hits” in two different sets of nodes is greater than 0. In such cases, you would create a rule that includes one or more set components.

Set components are a category of components that represent sets of nodes. These components include collectors and set operators. Collectors let you restrict a node or node set’s quantity. Set operators let you specify a relationship between two or more set components (for example, they could be used to create a rule that checks that the total number of hits in two rule conditions is less than 1).



Using Set Operators to Specify Relationships Between Set Components

You can use set operators to create rule segments that collect the number of hits of a specific relationship between two set components. For exam-

ple, you could use set operators to create a rule segment that collects the number of hits of the pattern of nodes that are in either set component A, or in set component B.

Set operators are rule elements that indicate two things:

- Which two set components you want to represent a relationship between.
- What type of relationship you want to establish between the two associated set components.

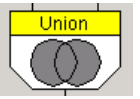
Because set operators establish relationships between set components, they must be connected to one set component (the set component that the set operator is attached to) and reference another set component (the “operand” of the set operator; i.e., the other set component that this set operator works with). Set operators can be used to establish four types of relationships:

- a union of two set components
- an intersection of two set components
- an exclusive-or relationship between two set components
- a difference between two set components

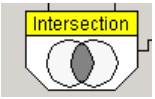
A union is the set of nodes that are:

- in the attached collector, or
- in the operand, or
- in both the attached collector and the operand.

For example, if A represents a set component that contains nodes xx, yy, and zz, and B represents a set component that contains nodes ww, xx, and yy, a union between A and B would match xx, yy, zz, and ww.



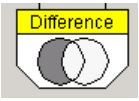
An intersection is the set of nodes that are in both the attached set component and the operand. For example, if A represents a set component that contains nodes xx, yy, and zz, and B represents a set component that contains nodes ww, xx, and yy, an intersection between A and B would match xx and yy.



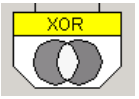
A difference is the set of nodes that are either:

- In the attached set component (the “left” side) but not in the operand (the “right” side), or
- In the operand (the “right” side), but not in the attached set component (the “left” side).

Thus, a right - left difference represents the nodes in the operand, but not the set operator, while a left - right difference represents the nodes that are in the set operator, but not in the operand. For example, if A represents a set component that contains nodes xx, yy, and zz, and B represents a set component that contains nodes ww, xx, and yy, an A-B difference between A and B would match zz, while a B-A difference would match ww.

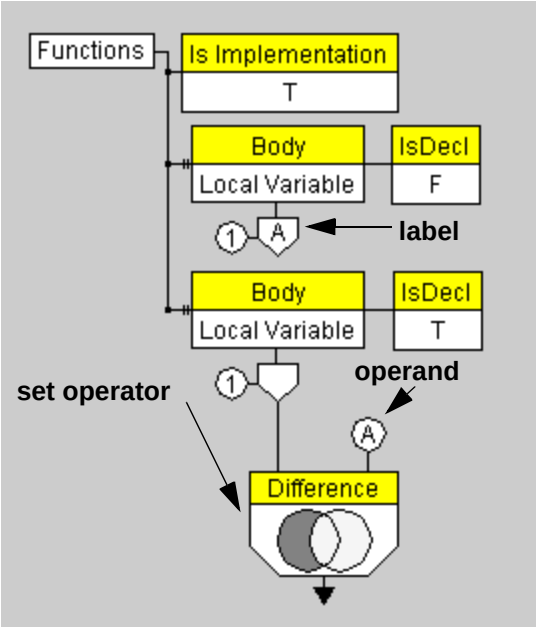


An XOR (exclusive-or) relationship is the set of nodes that are in either the attached set component or the operand, but not the nodes that are in both. For example, if A represents a set component that contains nodes xx, yy, and zz, and B represents a set component that contains nodes ww, xx, and yy, a XOR between A and B would match zz and ww.



To establish such relationships between two set components (X and Y), you would:

1. Attach a set operator to one set component (X) by right-clicking it and choosing **Set Operator> <desired relation>** from the short-cut menu that opens.
2. Attach a label to the other set component (Y).
3. Specify that you want Y to be the set operator's operand by selecting Y's label as the set component's operand value.



One common reason that you might use set operators is to create a rule that restricts the number of “hits” for a union, intersection, difference, or XOR node set.

To create a rule that counts and restricts the total number of hits that occur for a specific type of relationship between two node sets:

1. Create a rule condition that contains a set component (such as a collector).
2. Label the set component by right-clicking it, then choosing **Label> <label name>** from the shortcut menu that opens. Once you have done this, you can make this component an operand of a set operator.
3. Create another rule condition that contains a set component.

4. Right-click the newly-added set component, then choose **Create Set Operator> <desired type of relationship>** from the shortcut menu that opens. A set operator will then be attached to this set component.
5. Indicate the operand of this set operator by right-clicking the circle on the top right of the set operator and choosing the label of the first set component from the shortcut menu.
6. Restrict the number of "hits" allowed for the specified relationship by right-clicking the set operator, choosing **Count** from the shortcut menu, then (if necessary) modifying the value included in the count node.

Once you have added an output arrow to your rule and specified rule properties, your rule will be complete.

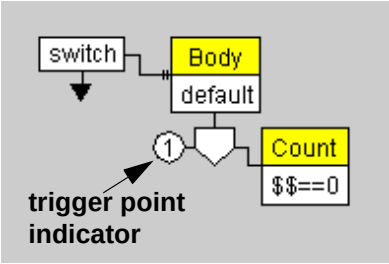
Customizing Counts With Collectors, Counts, and Trigger Points

About Trigger Points

RuleWizard's **Collector** and **Count** commands let you create a rule condition that restricts a node or node set's quantity. To create such a rule condition, you would first create a collector, then right-click the collector and choose **Count**. The collector keeps track of the number of times that the specified pattern is found; **Count** places a condition on what number of instances constitutes a rule violation. When a rule that restricts a node's quantity is enforced, the number of instances of the pattern are collected in the collector, the count is checked, and a violation is reported if the count falls within the parameters specified in the rule.

By specifying when CodeWizard should empty the collector, you can determine precisely how the count is determined. This is done through the use of "trigger points." Trigger points determine the node at which CodeWizard starts and stops counting the number of instances that occur. The correct trigger point number to use is determined by counting back from the node to which the collector is attached, to the node at which you want the collector emptied and a violation reported (if the specified pattern is found).

For example, consider the following example:



In this example, a trigger point of 1 indicates to empty the collector after *each switch statement* is searched. (The **switch** node is one node “back” from the **Body** node to which the collector and trigger point are attached). When enforced, this rule would, for each file, look for a switch statement, then look for default statements in that switch statement’s body. The number of expressions that met this criteria would be placed in the collector, the count would be checked, and a violation would be reported if the count were zero. When another switch statement was parsed, the collector would then be emptied, and this process would be repeated for that switch statement.

If you had a trigger point of 2 (the highest possible value in this example), CodeWizard would empty the collector only after the *entire file* was searched. The collector accumulates all values found from the file node (1 actual and one applied node (the file is an applied node) back from the node to which the trigger point and collector are attached), and the collector would not be emptied until a new file was parsed.

Creating a Trigger Point

When you create a collector, it is assigned a number 1 trigger point by default. To change the trigger point value, right-click the trigger point number and choose the desired value from the shortcut menu that opens.

Guidelines for using trigger points:

- To get the maximum value for a particular trigger point, count the number of nodes from the node with the attached collector to the top node of the rule, then add 1.
- You cannot place a trigger point on a node for which a direct check is performed.
- You cannot add an output arrow between the collector and the node/applied node that the trigger point number points to.

Determining the Output Type of a Set Component

If you place an output arrow on a set component, you will be asked to determine when the output “fires” (reports that the pattern has been violated). The available output options for set components are:

- **Hits Output:** Fires on each “hit”, or each node contained in the set. This type of output arrow is placed below the center of the set operator that it is attached to. When you choose this type of output, the violation output can contain fields of nodes contained in the set operator. For example, if you have a collector that contains variables in Java or C++ and you select a Hits Output, you can use an output message such as “Initialize all variables in constructor. Variable $name is not initialized.” (When this rule is actually violated, the $name variable will be replaced with the name of the uninitialized variable).
- **Trigger Output:** Fires once for the set when the set is triggered at the trigger point. This type of output arrow is placed below the left side of the set operator that it is attached to, or below the trigger point number. When you choose this type of output, the violation output can contain properties of the collector, but not properties of nodes contained in the set operator. Currently, the only variable you can use here is $count.

If your rule includes a collector, you can have your output include the number of “hits” that the collector has accumulated. To do this, enter COUNT(A) (where A is the label of the collector whose count you want reported) in the appropriate output message.

You can also have your output message list the items that a collector has accumulated. To do this, enter LIST(A) (where A is the label of the collector whose “list” you want reported) in the appropriate output message.

These two variables (COUNT and LIST) are called set references.

File Menu

The File menu contains the following commands:

- **Customize Preferences:** Opens the RuleWizard Preferences panel, which allows you to customize such RuleWizard options as rule view and rule file directory.
- **Save Preferences:** Saves the current RuleWizard Preferences.
- **Print:** Prints the contents of the current window. How well this works depends on your Java™ implementation.
- **Exit:** Closes RuleWizard. Any rules that were open will be saved as you exit.

Nodes Menu

The Nodes menu contains the following commands:

- **Properties:** Indicates which Node Dictionary you are currently using.
- **C:** This menu item check box is filled with red color as default. It indicates that all C-specific elements in the Node Dictionary are shown. When this check box is toggled to empty, RuleWizard will hide all C-specific elements in the Node Dictionary.
- **C++:** This menu item check box is filled with red color as default. It indicates that all C++-specific elements in the Node Dictionary are shown. When this check box is toggled to empty, RuleWizard will hide all C++-specific elements in the Node Dictionary.

Rule Menu

The Rule menu contains the following commands:

- **New Rule:** Closes the current rule and adds the first node of a new rule.
- **Open Rule:** Opens a saved rule.
- **Close:** Closes the current rule.
- **Save:** Saves the current rule.
- **Save As:** Saves the current rule; lets specify rule name and location.
- **Properties:** Allows you to describe the properties of the current rule, including Rule ID, Header, Severity, Author and Description fields.
- **Update Documentation:** Updates automatically generated documentation for custom rules. Rules documentation is accessible by choosing **Help> Rules Documentation**.

View Menu

The View menu contains the following commands:

- **Show/Hide status bar:** Displays/hides the status bar at the bottom of the GUI.
- **Show/Hide file viewer:** Displays/hides the file tab.

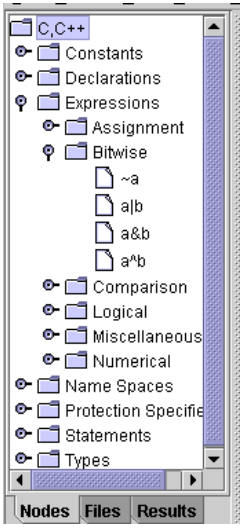
Help Menu

The Help menu contains the following commands:

- **RuleWizard Documentation:** Displays the RuleWizard User's Guide.
- **Rule Documentation:** Displays the documentation that RuleWizard automatically generates for custom rules. To update this documentation, choose **Rule> Update Documentation**.
- **About...:** Displays RuleWizard's version number.

Nodes Tab

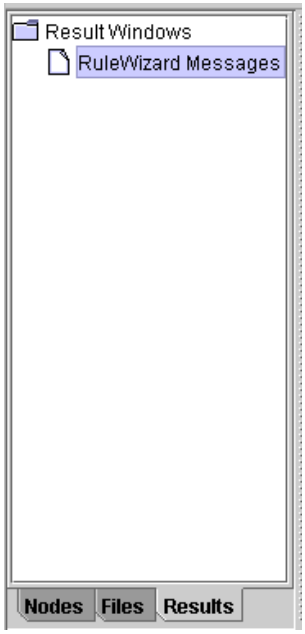
The Nodes tab contains the main elements of C and C++code. You can use the Nodes tab to start creating rules (by right-clicking the node that you want to be your parent rule node and choosing **Create Rule** from the shortcut menu that opens).



Results Tab

Clicking the Results tab opens the results tree, which displays all messages that RuleWizard has generated. To view the results tree, click the Results tab at the bottom of the left GUI pane. To view RuleWizard Messages, simply click on that category; all results will be displayed in the right GUI pane.

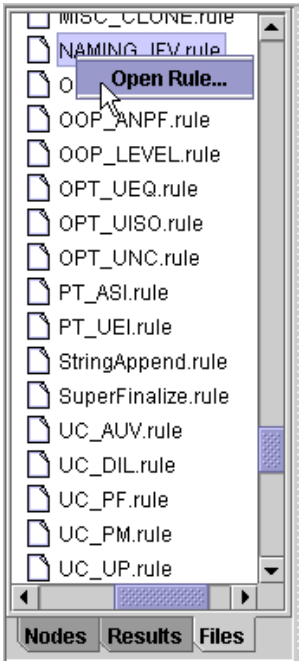
Clicking the RuleWizard Messages results category will allow you to access all messages that have appeared on the status bar. This feature provides you with a convenient way to go back and review a message that is no longer visible. To clear the RuleWizard Messages, right-click that category and then click **Clear** in the shortcut menu.



Files Tab

The Files tab displays the structure of your directories. You can use it to open rule files; to do this, right-click the rule file that you want to open, then choose **Open Rule** from the shortcut menu.

If you do not see the Files tab, you may enable it by choosing **View> Show File Viewer**.



Status Bar

The status bar displays RuleWizard messages, including tips on how to make the rule-in-progress valid. The color of the bar in the right side of the status bar indicates whether or not a rule is valid: a red bar indicates that the rule is not yet valid; a green bar indicates that the rule is valid. The messages in the status bar tell you how make an invalid rule valid.

Rule Properties Panel

The Rule Properties panel allows you to specify the following rule properties:

- Rule ID
- Header
- Severity
- Language Selection
- Author
- Description

To open the Rule Properties panel, choose **Rule> Properties**.

The Rule Properties panel has two tabs:

- **CodeWizard:** Determines properties that will be used within CodeWizard.
- **Info:** Lets you store general information about the rule.

CodeWizard Tab

The CodeWizard tab allows you to specify the following properties:

- **Rule ID:** Type the number that you want CodeWizard to assign to this rule. Each rule should have a unique Rule ID. When this rule is violated, CodeWizard will report a violation of rule "user [Rule ID]".

  Note: CodeWizard assigns Rule IDs from 100-699 to built-in user rules. If you assign your custom rule a Rule ID that is also assigned to a built-in rule, CodeWizard will report violations of both of these rules as violations of the same Rule ID. (For example, if you have two different rules whose Rule ID is 101, a violation of either rule will be flagged as a violation of user_101).

  You can prevent such confusing output by ensuring that every rule has a unique Rule ID.

- **Header:** Determines the name that CodeWizard assigns to this rule.
- **Severity:** Choose the severity category in which you want CodeWizard to classify the rule.

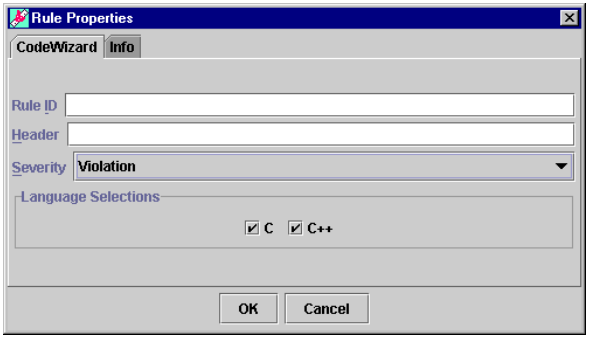
  CodeWizard classifies rules into groups based on the severity of violating the rule:

  - Severe Violation
  - Possible Severe Violation
  - Violation
  - Possible Violation
  - Informational

  The Severe Violation category should contain rules whose violation will *definitely* result in a bug. Each category below Severe Violation should contain rules whose violations have a progressively lower probability of resulting in an error.

  CodeWizard users can suppress rules according to their severity, so it is important to classify each error appropriately (If a violation of this rule will very likely lead to an error, give it a high severity; if you classify a severe rule as Informational, a violation of that rule may be overlooked by a user suppressing Informational violations.)

  Note: If you do not specify the rule's severity, it will be categorized as a Violation (the default setting).
- **Language Selections:** Determines which language(s) this rule applies to.



Info Tab

The Info tab allows you to specify the following properties:

- **Author:** Indicates the name of the rule's author.
- **Description:** Contains a description of this rule. This description will be displayed when you open a Message Window from Insra

RuleWizard Preferences Panel

The RuleWizard Preferences panel lets you specify RuleWizard options related to...

- How rules are displayed.
- Where rules are stored.
- How rule files are enabled.
- The web browser to be used.

To open the RuleWizard Preferences panel, select **File> Customize Preferences**.

The Rule Properties panel has three tabs:

- **View:** Determines how rules are displayed.
- **Rule Files:** Determines where rule files are stored.
- **Browser:** Sets browser-related options (such as browser used, browser commands, etc.).

View Tab

The View Tab allows you to specify the following preferences:

- **Pixel spacing between components**: Determines the space between rule nodes. Available options include:
    - **Horizontal:** Determines the horizontal spacing between rule nodes.
    - **Vertical:** Determines the vertical spacing between rule nodes.
    - **Reset to defaults:** Returns spacing settings to their default values.

Rule Files Tab

The Rule Files tab lets you configure the following preferences:

- **Rule File Directory:** Determines where your rule files are stored. You can either choose the default directory, or specify an alternate directory.
- **Rule file enabling:** Determines how rule files are enabled. Available options include:
    - Ask me whenever I save a rule that is not enabled.
    - Do not enable rules automatically; do not ask me.
    - Enable rules automatically; do not ask me.

Browser Tab

The Browser tab lets you configure the following preferences:

- **Browser:** Determines what browser RuleWizard uses.
- **Command:** Determines browser commands such as executable name and any arguments that you want RuleWizard to pass to that browser. If you select a browser name that is provided, RuleWizard will automatically fill in both **Executable** and **Arguments**. If you select **Other**, you can click **Browse** to navigate to the appropriate **Executable** settings. For **Arguments**, enter "%1".
- **Use DDE:** Determines whether or not Dynamic Data Exchange (DDE) lets programs share information. If you select Use DDE, the changes you make to files as you are using RuleWizard will automatically be applied to other programs that use those same files. **Use DDE** is selected by default and may not be disabled for the **Automatic** option in the **Browser** field. It is selected by default but may be disabled for **Netscape Navigator** and **Internet Explorer**. When **Other** is selected in the **Browser** field, **Use DDE** is disabled by default and may not be enabled.

RuleWizard Commands

When you right-click a rule node or other rule element, a shortcut menu containing all available commands for that node or element will open. Most commands are programming elements or concepts. The following commands are unique to RuleWizard:

- **Create Collector:** Adds a collector, displayed as a pentagon, to the selected rule node. The collector allows you to place numerical stipulations on the outcome of a rule. Works with the **Count** command (available by right-clicking the collector). The collector keeps track of the number of times a pattern is found; **Count** places a condition on what number of instances constitutes a rule violation.
- **Create Output:** Adds an output arrow to the selected rule node. The output arrow is the essential closing to any rule. An output arrow tells RuleWizard to report an error if the specified pattern is found; if no output arrow is included, no violations of this rule can be reported. The placement of the arrow determines what line number is reported in the error message. For example, if you have a rule with nodes A, B, and C and you attach the output arrow to node C, the line number will reference the line where C occurs; if you attach the output to node A, the line number will reference the line where A occurs.

  If you are adding an output arrow to a set component (a collector or set operator), you must choose between the following two types of output arrows:

  - **Hits Output:** Choose this output to indicate that:
    - One output message should be reported each time that the attached node is matched
    - The output message should reference the line number of the node to which the collector is attached. For example, if you have a collector that contains variables in Java or C++ and you select a Hits Output, you can use an output message such as “Initialize all variables in constructor. Variable $name is not initialized.” (When this

rule is actually violated, the $name variable will be replaced with the name of the uninitialized variable).

When you choose this type of output, the violation output can contain variables (such as “$tag”, or “$value”) for nodes contained in the set collector or operator. For example, if you have a collector that contains variables in Java or C++ and you select a Hits Output, you can use an output message such as “Initialize all variables in constructor. Variable $name is not initialized.” (When this rule is actually violated, the $name variable will be replaced with the name of the uninitialized variable).

- **Trigger Output:** Choose this output to indicate that:
  - One output message should be reported each time that the trigger point is matched (for example, if the trigger point references a file, one message will be reported for each file).
  - The output message’s line number should reference the line number of the trigger point node. This type of output arrow is placed below the center of the set operator that it is attached to.

    When you choose this type of output, the violation output can contain properties of the collector, but not properties of nodes contained in the set operator. Currently, the only variable you can use here is “$count”.

- **Indirect/Direct Check:** When an indirect check is performed, CodeWizard searches for the specified condition in all nodes that have the specified relationship to the given node.
  When a direct check is performed, CodeWizard searches for the specified condition only in the first node that has the specified relationship to the given node.

- **Create Set Operator:** Creates a set operator. Set operators are used to specify relationships between set components.

  Available options include:

  - **Union:** The set of nodes that are in either the attached set component or the operand, including the nodes that are in both.
  - **Intersection:** The set of nodes that are in both the attached set component and the operand.
  - **Difference:** The set of nodes that are in the attached set component but not in the operand, or the set of nodes that are in the operand but not in the attached set component.
    - **Left minus Right:** The set of nodes that are in the attached set component (to the left of the set operator), but which are not in the operand (as indicated in the circle attached to the right of the set operator).
    - **Right minus Left:** The set of nodes that are in the operand (as indicated in the circle attached to the right of the set operator), but which are not in the attached set component (to the left of the set operator).
  - **XOR:** The set of nodes that are either in the attached set component, the operand, but not in both.

  For information on set components and set operators, see “Working With Node Sets” on page 32.

- **Label:** Labels a set component so that it can be used as the operand of a set operator.

RuleWizard commands will be displayed at the top of the shortcut menu; options that pertain to programming elements and concepts are displayed below the menu's line.

Each non-RuleWizard command in the shortcut menu is followed by a symbol that describes the function of the item:

[...] indicates an item which can have either a direct or indirect check.

(S) indicates an item that takes a string input.

[T/F] indicates an item that lets you toggle between true and false inputs.

[M] indicates an item that lets you choose between predetermined input options.

(#) indicates an item that takes a numerical input.

***** indicates a property/node that is available from more than one command. For example, if you can choose the **Body** property from more than one of the available commands, the **Body** command will contain an asterisk.

The commands available depend on which node or rule element was selected. Some of these commands that you may not be familiar with include:

- **Count:** Lets you create a rule condition that restricts a node's quantity. Must be used in conjunction with the **Create Collector** command (First, create a collector, then right-click the collector and choose **Count**). The collector keeps track of the number of times a pattern is found; **Count** places a condition on what number of instances constitutes a rule violation. For information on determining exactly how counts are calculated, see "Working With Node Sets" on page 32.
- **Body:** Lets you create a rule condition about the code element that is a subnode of the parent node. (The body of Node A returns a "body" that is A's subnode; the exact definition of the "body" depends on the node itself).
- **Context:** Lets you create a rule condition about the code element that contains the parent node. (The context of Node A returns the node that contains Node A). For example, if you wanted to create a rule that said "always put X inside of Y," you would create a parent node for X, use Context to attach Y, create a collector, then use Count to specify that a count of $$==0 constitutes a violation.

  *Note: Some rules can use only **Body**, some can use only **Context**, some can use neither, and some can use both. If you have a choice, choose **Body** because **Body** will result in better performance than **Context**. In many-- but not all-- cases, **Body** and*

Context *are inverse operations. For example, an expression can be in the context of an statement, but be contained in the condition (rather than the body) of the statement.*

- **Condition:** Lets you create a rule condition about the parent node's condition statement. (The condition of Node A returns node A's condition statement).

Expressions and Regular Expressions

Expressions

Expressions are used to match values; $$ is used with expressions to indicate a variable. You can enter expressions in the Modify Expression window. A few examples of valid expressions that you could enter in this window include:

Expression	Matches	Example
$$==n	a value equal to n	$$==1 matches values equal to 1
$$<n	a value less than n	$$<100 matches values less than 100
$$>n	a value greater than n	$$>100 matches values greater than 100
$$<=n	a value less than or equal to n	$$<=550 matches values less than or equal to 550
$$>=n	a value greater than or equal to n	$$=>1 matches values greater than or equal to 1

Regular Expressions

Regular expressions are used to match strings. A regular expression is similar to a Perl expression. You can enter regular expressions in **Regexp** fields of Modify String windows. Here are some guidelines for entering regular expressions:

character/ metacharacter	**Matches**	**Examples**
anystring	an occurrence of the string “anystring”	“soft” matches parasoft, software, soften, etc.
.	exactly one non-null character	“.at” matches hat, cat, bat, fat, etc., but not “at” w...ing matches webking, working, but not “what a king” or “wing”
?	0 or 1 occurrences of the preceding character	“j?test” matches either jtest or test
*	0 or more occurrences of preceding character	“a*soft” matches asoft, or aaaaasoft; “.*ing” matches webking, waning, wing, what was that thing
+	1 or more occurrences of the preceding character	“a+soft” matches aaaaasoft, or aaasoft, but not asoft

character/ metacharacter	Matches	Examples
[]	matches one occurrence of any character inside the brackets; ^ inverts the brackets meta-character	“[cpy]up” matches cup, pup, or yup “rule0[1-4]” matches rule01, rule02, rule 03, rule 04 “[^ch]at” matches all 3 letter words ending with “at” except for cat and hat.
[A-Z]	any uppercase letters from A to Z	“[A-Z]” matches any uppercase letter from A to Z
[a-z]	any lowercase letters from a -z	“[a-z]” matches any lowercase letter from a-z
[0-9]	any integer from 0 to 9	“rule[0-9]” matches any expression that begins with “rule” and ends with an integer
(?i)	ignore case	“(?i)ParaSoft” matches ParaSoft, PARASOFT, parasoft or paraSOFT
{}	like *, but the string it matches must be of the length specified in the braces	“a{2}” matches aaa “a{3,}” matches at least 3 occurrences of the preceding character (aaaaaa, or aaaaaaaa, but not aa “a {2,5}” matches between 2 and 5 occurrences of the preceding character (aaa, aaaaaa, but not aa or aaaaaaaaaaaaaaaaa)

character/ metacharacter	Matches	Examples
|	matches the string before the “|”, the string after the “|”, or both	“rulewizard|codewizard” matches rulewizard, codewizard, or both

Additional Tips

- ^ indicates the beginning of a string in parentheses; $ indicates the end of a string in parentheses. Thus, to get an exact match for a string, use the format ^(STRING)$. For example ^(soft)$ would only flag “soft”.
- If you want CodeWizard to report an error if the expression is detected, leave the Regexp window’s **Negate** check box empty.
- If you want CodeWizard to report an error if the expression is not detected, check the Regexp window’s **Negate** check box.
- Regular expressions searches are case sensitive by default.
- When using regular expressions, “\" is an escape character that you can use to match a "." , "*", or another character that has a non-literal meaning.

Available Rule Nodes

The nodes that you can use to create your rule are listed below, in the order in which they appear in a fully expanded node tree.

Constants

Constant value of any type.

bool Constant

A constant value of true or false.

enum Constant

An enum identifier declared in body of enum declaration.

Example:

```
enum E { A, B, C};          // enum Constant: A, B, C
```

Integer Constant

Constant integer expression.

Example:

```
int i = 10;               // Integer Constant, Value is 10
```

Real Constant

Constant expression which is a real number.

Example:

```
double d = 34.75;        // Real Constant, Representation is "34.75"
```

String Constant

Constant string expression

Example:

```
char *pc = "John Doe";     // String Constant, Value is "John Doe"
```

Declarations

Nodes representing various declarations in user code.

Some declaration nodes can be accessed from both their declaration and expressions which reference that declaration. In these cases, the IsDecl property returns a boolean, allowing users to distinguish between the actual declaration and references to that declaration.

Friend

friend function declaration.

Example:

```
void friendFunc();
class Foo {
public:
    friend void friendFunc();     // Friend
};
```

Functions

Function declarations.

Global Function

Global function declaration.

Example:

```
void globalFunc1();       // Global Function, IsDecl is true
                          // Is Implementation is false
void globalFunc2() { }    // Global Function, IsDecl is true
                          // Is Implementation is true
```

Member Function

Member function declaration.

Example:

```
class Foo {
public:
   void func();             // Member Function
};
```

Parameter

Function parameter declaration.

Example:

```
void func(int i);          // i is parameter whose type is int
```

Template Parameter

Template parameter declaration.

Example:

```
class <class T>
void func(T t) {}      // Template Parameter
```

Variables

Variables

Global Variable

Global Variable.

Example:

```
int global;            // Global Variable, IsDecl is true
global = 3;            // Global Variable, IsDecl is false
```

Local Variable

Local Variable.

Example:

```
void func() {
   int local;           // Local Variable, IsDecl is true
   local = 3;           // Local Variable, IsDecl is false
}
```

Member Variable

Member Variable.

Example:

```
class Foo {
public:
  Foo() : member(0) { }  // Member Variable, IsDecl is false
private:
  int member;            // Member Variable, IsDecl is true
};
```

Expressions

Includes all types of expression nodes.

Assignment

Includes all expression nodes used for assignment.

a=b

Assignment operator expression.

Example:

```
int i = 1;
int j = 0;
j = i;                   // Assignment a=b
```

a+=b

Plus-equals operator expression.

Example:

```
int i = 1;
int j = 0;
j += i;                 // Plus-equals a+=b
```

a-=b

Minus-equals operator expression.

Example:

```
int i = 1;
int j = 0;
j -= i;                 // Minus-equals a-=b
```

a/=b

Divide-equals operator expression.

Example:

```
int i = 4;
int j = 2;
j /= i;                 // Divide-equals a/=b
```

a*=b

Multiply-equals operator expression.

Example:

```
int i = 2;
int j = 3;
j *= i;                 // Multiply-equals a*=b
```

a%=b

Mod-equals operator expression.

Example:

```
int i = 2;
int j = 3;
j %= i;                 // Mod-equals a%=b
```

a&=b

Bitwise-and-equals operator expression.

Example:

```
int i = 2;
int j = 3;
j &= i;                 // Bitwise-and-equals a%=b
```

a^=b

Bitwise-xor-equals operator expression.

Example:

```
int i = 2;
int j = 3;
j ^= i;                 // Bitwise-xor-equals a%=b
```

a|=b

Bitwise-or-equals operator expression.

Example:

```
int i = 2;
int j = 3;
j |= i;                 // Bitwise-or-equals a%=b
```

a<<=b

Left-shift-equals operator expression.

Example:

```
int i = 2;
int j = 3;
j <<= i;                // Left-shift-equals a%=b
```

a>>=b

Right-shift-equals operator expression.

Example:

```
int i = 2;
int j = 3;
j >>= i;                // Right-shift-equals a%=b
```

--a

Predecrement expression.

Example:

```
int i = 1;
--i;                    // Predecrement --a
```

++a

Preincrement expression.

Example:

```
int i = 1;
++i;                    // Preincrement ++a
```

a--

Postdecrement expression.

Example:

```
int i = 1;
i--;                    // Postdecrement a--
```

a++

Postincrement expression.

Example:

```
int i = 1;
```

```
i++;                    // Postincrement a++
```

Bitwise

Expressions involving non-assignment bitwise operators.

~a

Bitwise 'not' expression.

Example:

```
int a = ~3;             // Bitwise 'not' ~a
```

a|b

Bitwise 'or' expression.

Example:

```
int x = 73, y = 0;
y = x | 4;             // Bitwise 'or' a|b
```

a&b

Bitwise 'and' expression.

Example:

```
int z = 73, y = 0;
y = z & 0x0f;             // Bitwise 'and' a&b
```

a^b

Bitwise XOR expression.

Example:

```
int z = 73, y = 0;
y = z | 0x0f;             // Bitwise XOR a^b
```

Comparison

Expressions which compare values.

a==b

Equality operator expression

Example:

```
bool func(int x) {
    if (x==3) {        // a==b
        return false;
    }
    return true
}
```

a!=b

Not-equal operator expression.

Example:

```
bool func(int x) {
    if (x!=3) {        // a!=b
        return false;
    }
    return true
}
```

a<b

'Less than' expression.

Example:

```
bool func(int x) {
    if (x<3) {        // a<b
        return false;
    }
    return true
}
```

a<=b

'Less than or equals' expression.

Example:

```
bool func(int x) {
    if (x <= 3) {        // a<=b
        return false;
    }
    return true
}
```

a>b

'Greater than' expression.

Example:

```
bool func(int x) {
    if (x > 3) {        // a>b
        return false;
    }
    return true
}
```

a>=b

'Greater than or equals' expression

Example:

```
bool func(int x) {
    if (x >= 3) {        // a>=b
        return false;
    }
    return true
}
```

Logical

Comparison expressions expressions involving logical operators.

!a

Logical 'not' expression.

Example:

```
bool func(int x) {
    if (!x) {        // !a
        return false;
    }
    return true
}
```

a&&b

Logical 'and' expression.

Example:

```
bool func(int x, int y) {
    if (x && y) {        // a&&b
        return false;
    }
    return true
}
```

a||b

Logical 'or' expression.

Example:

```
bool func(int x, int y) {
    if (x || y) {        // a||b
        return false;
    }
    return true
}
```

Miscellaneous

Miscellaneous expressions which don't fit in other categories.

ellipses (...)

Ellipses expression, used for catch parameter.

Example:

```
void func() {
  try {
  } catch(...) {}               // catch parameter is (...)
}
```

a.b

Dot operator expression.

Example:

```
class Foo {
public:
  void func();
  int zzz;
};
int bar() {
  Foo f;
  f.func();                    // a.b
```

```
  return f.zzz;                // a.b
}
```

a::b

Scope reference expression.

Example:

```
class Foo {
public:
   static void func();
};
void bar() {
   Foo::func();                // a::b
}
```

&a

Address of' expression.

Example:

```
int i = 0;
int *j = &i;                 // &a
```

*a

Dereference expression.

Example:

```
int *i = new int();
*i = 5;                      // *a
```

a.*b

Dot-star expression, used in pointers to member functions.

Example:

```
class Foo {
public:
   void func();
};
typedef void(Foo::*mpf)();
void bar(Foo f) {
  mpf ampf = &Foo::func;
  (f.*ampf)();               // a.*b
```

```
}
```

a->*b

Arrow-star expression, used in pointers to member function.

Example:

```
class Foo {
public:
   void func();
};
typedef void(Foo::*mpf)();
void bar(Foo *f) {
  mpf ampf = &Foo::func;
  (f->*ampf)();                 // a->*b
}
```

a[b]

Array reference.

```
char array[] = "John Doe";
char c = array[0];                // array reference a[b]
```

a,b

Compound statement list.

Example:

```
void func() {
    for (int i = 0; i >=0, i < 10; i++) {  // a,b in for condition
    }
}
```

a(b)

Function call expression.

Example:

```
void foo(int i) {}
void bar(int x) {
    foo(x);                   // Function call a(b).
                              // Left Hand Side is Function foo,
                              // Right Hand Side is Parameter x
}
```

a?b:c

Ternary operator expression.

Example

```
bool func(int x) {
  return x ? true : false;    // Ternary operator a?b:c
}
```

typeid

typeid operator expression, used for RTTI.

Example:

```
class Shape {};
void f(Shape& r) {
   typeid(r);               // typeid
}
```

sizeof

sizeof operator expression.

Example:

```
int *i = malloc(20*sizeof(int));  // sizeof
```

new

new operator expression.

Example:

```
int *i = new int[20];            // new
```

delete

delete operator expression.

Example:

```
int *i = new int();
delete(i);                      // delete
```

throw

throw operator expression.

Example:

```
void func() {
   throw;                        // throw
}
```

asm

Inlined assembly expression.

Example:

```
void func() {
   asm(mov eax, ebp);            // asm
}
```

Cast

Expression which casts from one type to another.

Kind of cast indicated by "Kind", which include the following:

- normal
- implicit
- dynamic_cast
- const_cast
- static_cast
- reinterpret_cast

Example:

```
class Base {
public:
   virtual ~Base();
};
class Derived : public Base {};
void func() {
    char c = 'c';
    int i = (int)c;                          // normal (C-style) cast
    float f = i;                                // implicit cast
    Base * pb = new Derived();
    Derived* pd = dynamic_cast<Derived *>(pb); // dynamic_cast
    const char * pname = "John Doe";
    char * pc = const_cast<char *>(pname);     // const_cast
    c = static_cast<char>(i);                  // static cast
```

```
    int *pi = reinterpret_cast<int *>(pc);     // reinterpret cast
}
```

Overloaded

An expression using an overloaded operator.

Example:

```
#include <iostream.h>
void func()
{
   cout << "Hello, how old are you?"   // Overloaded <<
        << endl;                      // Overloaded <<
   int age;
   cin >> age;                        // Overloaded >>
}
```

ab (string concat)

Expression of strings being concatenated.

Example:

```
char *pc = "ab" "cd";             // ab (string concat)
```

Initializer List

Expression containing initializations, used in member functions.

Example:

```
class Foo {
public:
   Foo() _x(0), _y(0) {}          // Initializer list.
private:
   int _x;
   int _y;
};
```

Numerical

Numerical expressions which are not assignments.

+a

Positive signed expression.

Example:

```
int func(int num) {
   return +num;                  // +a
}
```

-a

Negative signed expression.

Example:

```
int func(int num) {
   return -num;                  // -a
}
```

a+b

Addition expression.

Example:

```
int j = 1;
int k = 3 + j;           // Addition a+b
```

a-b

Subtraction expression.

Example:

```
int j = 1;
int k = 3 - j;           // Subtraction a-b
```

a*b

Multiplication expression.

Example:

```
int j = 1;
int k = 3 * j;           // Multiplication a*b
```

a/b

Division expression.

Example:

```
int j = 2;
int k = 5 / j;           // Division a/b
```

a%b

Modulo expression.

Example:

```
int j = 2;
int k = 5 % j;           // Mod a%b
```

a<<b

Left-shift expression.

Example:

```
int i = 1, j = 1, k = 1;
k = i << j;              // Left-shift a>>b
```

a>>b

Right-shift expression.

Example:

```
int i = 1, j = 1, k = 1;
k = i >> j;              // Right-shift a>>b
```

Name Spaces

Nodes involving the use of namespaces.

namespace

Namespace declaration

Example:

```
namespace ns {        // Name Space
  class Foo {}
};
```

using

Use of 'using' namespace keyword.

Example:

```
#include <vector>
using namespace std;   // using
```

Miscellaneous

Various nodes not fitting in other categories.

File

File containing source code.

Used for rules about relationships of nodes within a files.

Protection Specifiers

Nodes which specify access accessibility.

public

Public accessibility.

Example:

```
class Foo {
public:              // public
   Foo();
};
```

protected

Protected accessibility.

Example:

```
class Foo {
protected:           // protected
   Foo();
};
```

private

Private accessibility.

Example:

```
class Foo {
private:            // private
   Foo();
};
```

Statements

General node for all types of statements.

break

Break statement.

Example:

```
bool bar();
void func() {
   while(1) {
      if (bar()) {
          break;    // break
      }
   }
}
```

case

Case statement.

Example:

```
void func(int i) {
    switch(i) {
    case 1: { break; }  // case
    case 2: { break; }  // case
    }
}
```

catch

Catch statement, used for exception handling.

Example:

```
void bar();
void func(int i) {
    try {
       bar();
    } catch (...) {      // catch
    }
}
```

continue

Continue statement, used in loops.

Example:

```
bool bar(int);
void func(int i) {
    while(i--) {
       if (bar(i)) {
          continue;     // continue
       }
    }
}
```

default

Default statement, used inside switch statement.

Example:

```
void func(int i) {
    switch(i) {
    case 1: { break; }
    case 2: { break; }
    default: { break; }   // default
    }
}
```

do while

do while loop statement.

Example:

```
void func(int i) {
    do {                  // do while
        i--;
    } while (i);
}
```

for

for loop statement.

Example:

```
void bar();
void func() {
   for (int i = 0; i < 10; i++) {  // for
      bar();
   }
}
```

goto

goto statement.

Example:

```
void func(bool done) {
   if (done) {
      goto end;             // goto
   }
   end:
}
```

if

if statement.

Example:

```
void func(bool done) {
   if (done) {             // if
       return;
   }
}
```

return

return statement.

Example:

```
int func() {
   return 0;               // return
}
```

switch

switch statement.

Example:

```
void func(int i) {
    switch(i) {           // switch
    case 1: { break; }
    case 2: { break; }
    default: { break; }
    }
}
```

try

try statement.

Example

```
void bar();
void func(int i) {
    try {           // try
        bar();
    } catch (...) {
    }
}
```

while

while statement.

Example:

```
void func(int i) {
   while (i--) {   // while
   }
}
```

Block

Block statement. Normally contains other statements.

Begins and ends with curly braces.

Example:

```
void func() {         // block
}
```

Simple

A statement which does not fall into any other statement category.

The simple statement's body usually contains an expression.

Example:

```
void func(int & i) {
    i++;             // Simple statement
}
```

Types

All types which can be used for variable declaration.

Complex

Non-primitive types.

Array

Array type.

Example:

```
char buf[1024];     // buf is Variable of Type Array or Type char.
```

Builtin

Non-primitive type builtin for a given compiler.

Class

User-defined class type.

Example:

```
class Foo { };       // class
```

Enum

User-defined enum type.

Example:

```
enum E {};            // Enum type
```

Function

Function type. Used for function pointers.

Example:

```
void foo(void (*ptr)(int)) {   // ptr is Pointer of Type Function
    ptr(3);
}
```

Reference

Reference type. Used for passing by reference.

Example:

```
void func(int &i) {  // i is Parameter of Type Reference to Type int
   i = 3;
}
```

Struct

Struct type.

Example:

```
struct Foo { };      // struct
```

Typedef

typedef type.

Example:

```
typedef int INT;
INT x;            // x is Variable of Type Typedef to Type int.
```

Union

Union type.

Example:

```
union MyUnion {   // union
    int x;
    char *y;
};
```

Primitive

All primitive types.

bool

Primitive type bool.

Example:

```
bool b;          // b is Variable of Type bool
```

char

Primitive type char.

Example:

```
char c;          // c is Variable of Type char
```

wchar_t

Primitive type wchar_t.

Example:

```
wchar_t c;          // c is Variable of Type wchar_t
```

short

Primitive type short.

Example:

```
short x;          // x is Variable of Type short
```

int

Primitive type int.

Example:

```
int i;            // i is Variable of Type int
```

long

Primitive type long.

Example:

```
long x;            // x is Variable of Type long
```

float

Primitive type float.

Example:

```
float f;            // f is Variable of Type float
```

double

Primitive type double.

Example:

```
double d;           // d is Variable of Type double
```

long double

Primitive type long double.

Example:

```
long double d;      // d is Variable of Type long double
```

void

Primitive type void. Used for void pointers.

Example:

```
void * p;           // p is Variable of Type pointer to type void.
```

pointer

Pointer type.

Example:

```
char * name;       // name is Variable of Type pointer to type char.
```

Index

A

B

C

D

E

F

H

I

L

M

N

O

P

Q

R

S

T

U

V

X